Automatic Speech Recognition using the Kaldi Toolkit

Group 24 Team members :

1..Catherine sp.

2..Adarsh singh.

3..Kishan taparia

4..Akash kumar.

Contents

1 Abstract

This project explores the current technology available for *Automatic Speech Recognition* (ASR), the process of converting speech from a recorded audio signal to text [11]. The primary goal is to identify a toolkit for use in the construction of a personal assistant system, similar to Amazon's Alexa, but with a smaller and more targeted lexicon meant to increase accuracy. In particular, we explore the Kaldi Speech Recognition Toolkit, written in C++ and licensed under the Apache License v2.0, developed for use by speech recognition researchers [17]. This toolkit was chosen on the grounds of extensibility, minimal restrictive licensing, thorough documentation (including example scripts), and complete speech recognition system recipes. In this project, we explore the ASR process used in Kaldi (including feature extraction, GMMs, decoding graphs, etc.). With this foundation, we walk through three extensions of the Kaldi toolkit: (1) the **Digits** example, using 1500 audio recordings of the digits 0-9, (2) the **VoxForge** example[3] and (3) the **CMU AN4** alphanumeric example[2]. This project demonstrates that Kaldi can be extended in simple and complex situations and is flexible and easy to use in development. Given the results of this analysis, we conclude that Kaldi is a viable choice for future extension.

2 Project Goals

The goal of this project is to develop a prototype system for Automatic Speech Recognition (ASR; the process of converting speech from a recorded audio signal to text [11]) satisfying the following requirements:

1. Easy to develop and extend
2. Lightweight and minimal
3. Accurate and fast

First and foremost, this system must be easy to develop and extend. If we cannot work with the system (because it is incredibly esoteric and/or not welldocumented), it is virtually useless. Likewise, if we cannot extend the system with custom data (which is integral to our system design), the system will not work for our purposes.

In addition, this product must be a lightweight and minimal ASR system – we need to maximize accuracy for a small state space of input options (geared towards the client), using a small device and streamlined system. The user should be able to request something within a small state-space of options and receive feedback accordingly with high accuracy. For instance, if the client were using such a system in a car shop, requests such as "purchase a Chevy Malibu A/C Compressor" should be parsed and carried out (e.g., through Amazon's marketplace). We must minimize run time so that the system can be used without inconvenience (for instance, a response time over 10 seconds would be non-viable).

In order to completely explore Kaldi, we hope to do the following:

1. Outline the layout of Kaldi
   Installation
   Organization
   Sub-components of Kaldi
   Data preparation (using custom data) Decoding the results
2. Walk through several examples using the Kaldi Toolkit Introductory example: Using 300hrs of English Dataset.

The final result should be a well-rounded understanding of the Kaldi system.

3 Literature Review

3.1 What is Automatic Speech Recognition?

Automatic Speech Recognition (ASR) is "the process of converting speech from a recorded audio signal to text" [11]. The particular type of ASR we are interested in is the *personal assistant* ASR system. These types of systems are seen across households today, in products like Amazon's Alexa, and must be able to respond quickly, accurately and helpfully to a user. We are interested in the "understanding" component of this system – the part of the assistant that "understands" what the user is saying (by translating the query from speech to text) and searching its resources for a response.

The typical model for ASR can be found in Figure 1. We start with an audio waveform and extract a series of "features," representations of small frames of the speech function. These features, along with a pronunciation dictionary to match features to phones, can be used to generate acoustic models (the likelihood of an observed acoustic signal given a word sequence). The likelihood of an observed word sequence is derived from a language model.

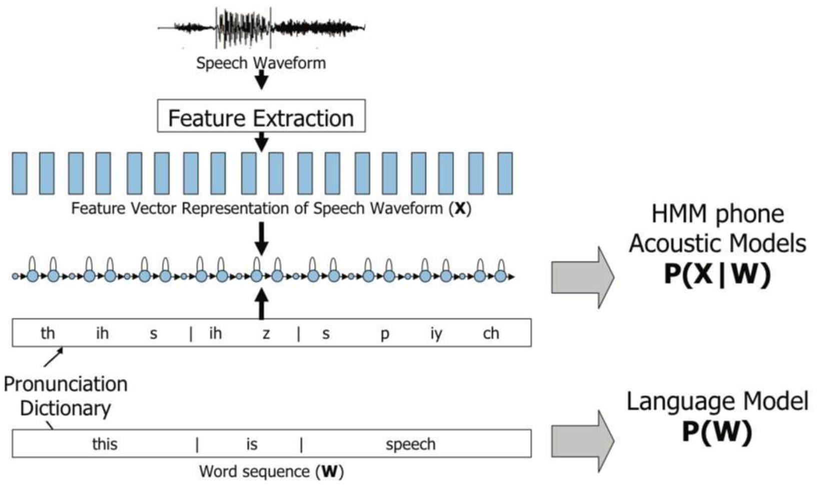


Figure 1: Typical process of Automatic Speech Recognition [12]

Automatic Speech Recognition is a complicated process and will not be completely outlined in this paper. Rather, we will explore the steps involved in interacting with an ASR system like Kaldi as a client. Should you be interested further in the theory, see the prior paper in this series [8] or other papers devoted to the theory of Automatic Speech Recognition (such as *Automatic Speech Recognition* by Gruhn et. al [11]).

3.2 What is Kaldi?

The Kaldi Speech Recognition Toolkit is a toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. It is intended for use by speech recognition researchers. At its inception in 2009, this toolkit was designed for “Low Development Cost, High Quality Speech Recognition.” Its founders felt that “a well-engineered, modern, general-purpose speech toolkit with an open license would be an asset to the speech-recognition community” [17]. Since its initial release, Kaldi has been maintained and developed largely by Daniel Povey (Researcher at Microsoft and John Hopkins University).

4 Kaldi: Automatic Speech Recognition Toolkit

4.1 Kaldi Layout

The general layout of the Kaldi Toolkit is displayed in Figure 2. It accepts a set of customizable audio data as input, along with accompanying language and acoustic data (see the *Data Preparation* section).

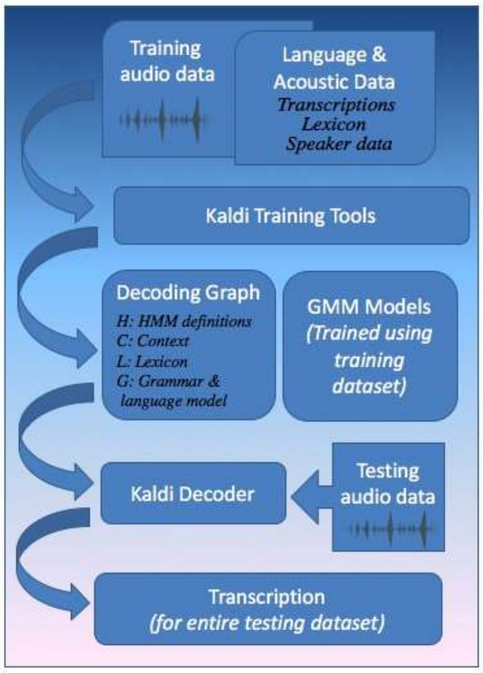


Figure 2: Layout of Kaldi Toolkit (based on NTNU diagram and Kaldi documentation) [17] [9]. Note that this diagram is hugely simplifying – optimizations and adjustments (e.g., using alignments) are not shown.

We may note that the input data is used to generate two main Kaldi components, the *decoding graph* and *final acoustic GMM*.

4.1.1 Decoding Graph

The first central element is a decoding graph (of the HCLG format; see Fig. 2). The **H** represents the Hidden Markov Model (HMM) structure, where an HMM is used to model a Markov Process (a stochastic process satisfying the Markov property of "memorylessness"). In this case, the structure map states to phonemes. The **C** represents contextual information about the phones (i.e., the articulation of a phone may change given surrounding phones). The **L** represents the lexicon, which maps each possible word to a set or several sets of phones.

Finally, the **G** represents the language model (or grammar) which estimates the probability of a given word sequence. Together, these components form a decoding graph which can be used to match a given input vector to a resulting transcription. The decoding graph for our Digits example, for instance, might look something like the network shown in Figure 3.

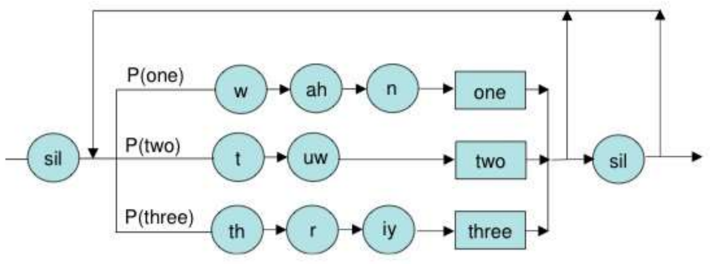


Figure 3: An example decoding graph with the words “one,” “two,” and “three” in the lexicon [12]

4.1.2 Acoustic GMMs

The second element is a final Gaussian Mixture Model (GMM). A GMM is a probabilistic model, in this case used to represent an acoustic output. Our final result in this process will be a series of GMMs matching to each state in our decoding network. Mapping the HMM structure to this GMM structure is done in the *run* script of each example [17].

It can be noted that we will primarily observe two types of GMM training: triphone and monophone. The first uses contextual information while the latter does not [10].

4.2 Decoding

Together, these pieces (*HCLG.fst* and *final.mdl*) can be fed into the decoder, along with testing features to produce transcriptions [17]. During the *run* process, the system will generate a series of transcriptions, documented in the decoding logs, which can be compared to the expected results manually or via the generated word error rate files.

4.3 Reader Caveat

As users of Kaldi, rather than true developers of Kaldi, we will focus on the start and end points of this flow, rather than the mechanics of the Kaldi training algorithms (if you have background in GMMs, decoding graphs, etc., the Kaldi documentation may be of interest to you [17]). Of primary interest to us are the customizable input (discussed in *Data Preparation*) and the decoding results.

4.4 Organization

The relevant Kaldi directories are organized in the following fashion:

1. *egs*: A series of example scripts allowing you to quickly build ASR systems for over 30 popular speech corpora (documentation is attached for each project)
2. *misc*: Additional tools and supplies, not needed for proper Kaldi functionality
3. *src*: Kaldi source code
4. *tools*: Useful components and external tools

We will be working in the *egs* folder, where all of the Kaldi extensions are housed. We will also use some of the scripts in the *tools* folder, which help with installation.

4.5 Installation

Kaldi is housed on Github, so installation is as easy as cloning the project, using the below command:

```
git clone https://github.com/kaldi-asr/kaldi.git kaldi --origin ...
        upstream
cd kaldi
```

To retrieve any new updates, users need only pull from this repo and refresh their project.

Actually running Kaldi will require building the project – this can be accomplished by following the README instructions and using the relevant Makefiles.

4.6 Data Preparation

Data preparation is the most relevant component of the Kaldi layout to this analysis. Because we seek to feed in customized data, we must understand the requirements of the system.

In each extension, we have to define:

1. Audio data (training and testing)
2. Acoustic data spk2gender: [speakerID] [gender] wav.scp: [utteranceID] [file path] text: [utteranceID] [transcription] utt2spk: [utteranceID] [speakerID] corpus.txt: [transcription]
3. Language data

   lexicon.txt: [word] [phone(s)] nonsilence phones.txt:

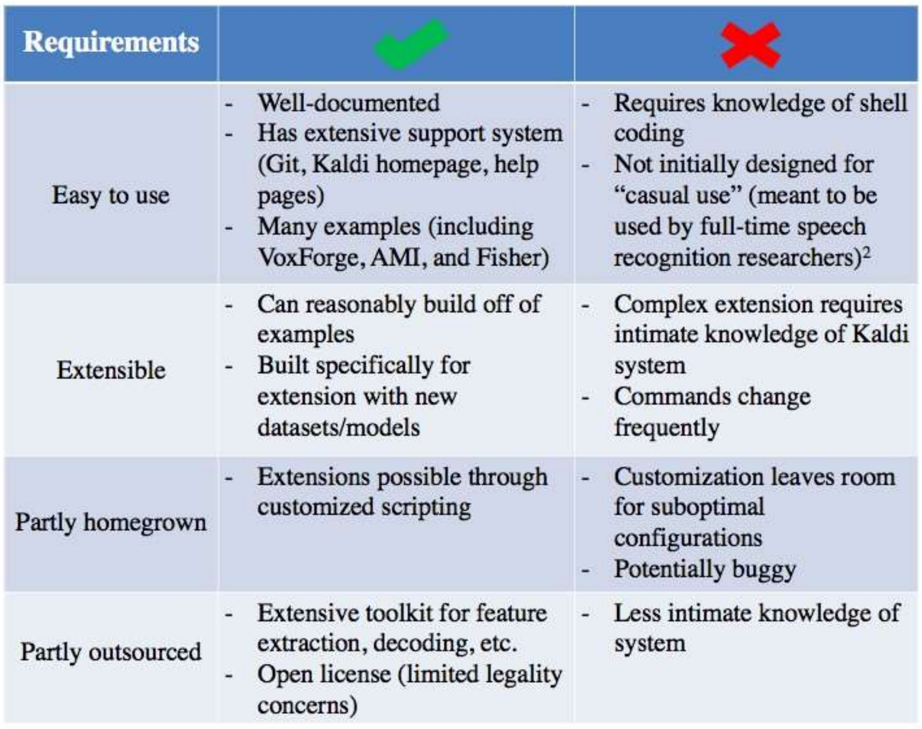
   [phone] silence phones.txt: [phone] optional silence.txt:

   [phone]
4. (Optional) Configuration
5. (Optional) Language model toolkit

We will see in the examples how such files may be manually or automatically generated.

5 Initial Assessment of Kaldi

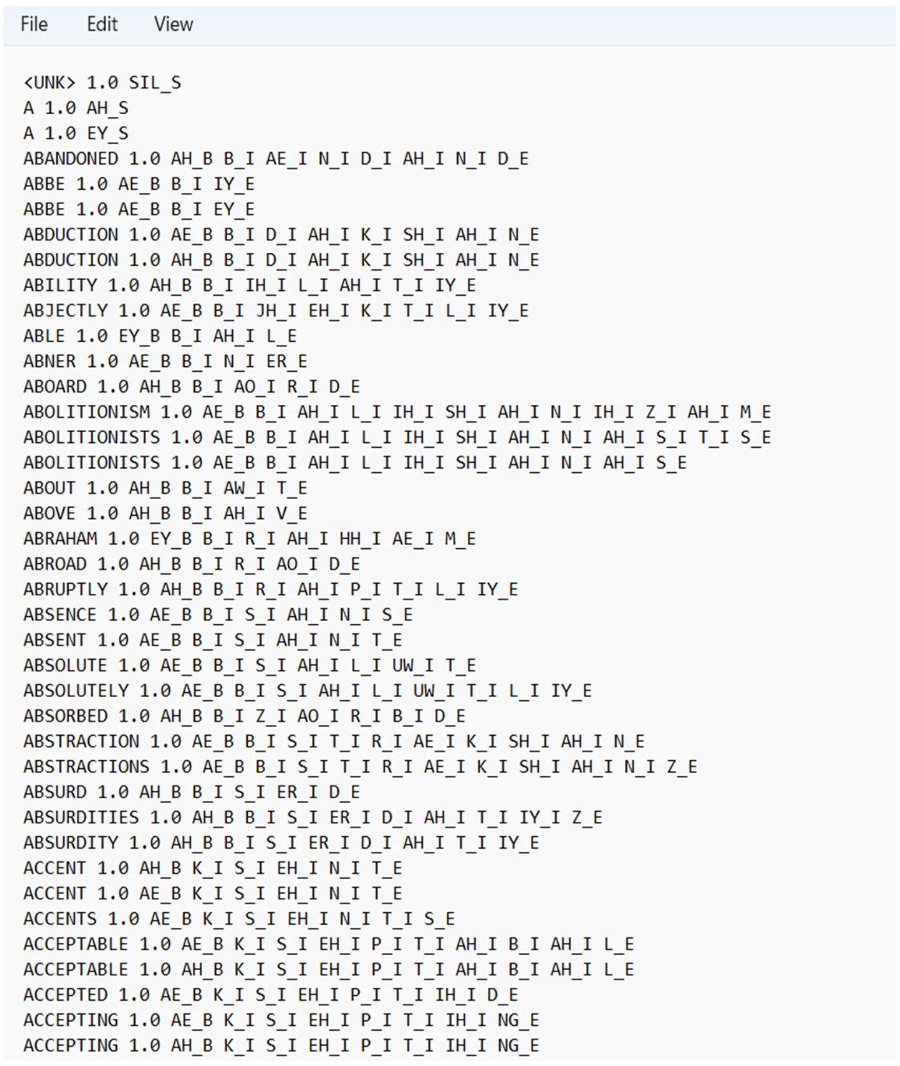
An initial assessment of Kaldi (see Figure 4) reveals it to be a viable system for the desired product. Kaldi includes a variety of utility scripts, including functionalities such as feature extraction, data preparation, transition modeling, construction of decoding graphs, and acoustic modelling. Extensions of Kaldi can incorporate custom training and testing data and use the corresponding lexicon. These extensions can still utilize the provided scripts, substituting in various decoding types, language models, etc.



Requirements	✓	✗
Easy to use	- Well-documented - Has extensive support system (Git, Kaldi homepage, help pages) - Many examples (including VoxForge, AMI, and Fisher)	- Requires knowledge of shell coding - Not initially designed for "casual use" (meant to be used by full-time speech recognition researchers)[2]
Extensible	- Can reasonably build off of examples - Built specifically for extension with new datasets/models	- Complex extension requires intimate knowledge of Kaldi system - Commands change frequently
Partly homegrown	- Extensions possible through customized scripting	- Customization leaves room for suboptimal configurations - Potentially buggy
Partly outsourced	- Extensive toolkit for feature extraction, decoding, etc. - Open license (limited legality concerns)	- Less intimate knowledge of system

Figure 4: Assessing the viability of Kaldi (note that speed was not considered in this analysis) [17] [9].

6 Digits Example



```
<UNK> 1.0 SIL_S
A 1.0 AH_S
A 1.0 EY_S
ABANDONED 1.0 AH_B B_I AE_I N_I D_I AH_I N_I D_E
ABBE 1.0 AE_B B_I IY_E
ABBE 1.0 AE_B B_I EY_E
ABDUCTION 1.0 AE_B B_I D_I AH_I K_I SH_I AH_I N_E
ABDUCTION 1.0 AH_B B_I D_I AH_I K_I SH_I AH_I N_E
ABILITY 1.0 AH_B B_I IH_I L_I AH_I T_I IY_E
ABJECTLY 1.0 AE_B B_I JH_I EH_I K_I T_I L_I IY_E
ABLE 1.0 EY_B B_I AH_I L_E
ABNER 1.0 AE_B B_I N_I ER_E
ABOARD 1.0 AH_B B_I AO_I R_I D_E
ABOLITIONISM 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I IH_I Z_I AH_I M_E
ABOLITIONISTS 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I AH_I S_I T_I S_E
ABOLITIONISTS 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I AH_I S_E
ABOUT 1.0 AH_B B_I AW_I T_E
ABOVE 1.0 AH_B B_I AH_I V_E
ABRAHAM 1.0 EY_B B_I R_I AH_I HH_I AE_I M_E
ABROAD 1.0 AH_B B_I R_I AO_I D_E
ABRUPTLY 1.0 AH_B B_I R_I AH_I P_I T_I L_I IY_E
ABSENCE 1.0 AE_B B_I S_I AH_I N_I S_E
ABSENT 1.0 AE_B B_I S_I AH_I N_I T_E
ABSOLUTE 1.0 AE_B B_I S_I AH_I L_I UW_I T_E
ABSOLUTELY 1.0 AE_B B_I S_I AH_I L_I UW_I T_I L_I IY_E
ABSORBED 1.0 AH_B B_I Z_I AO_I R_I B_I D_E
ABSTRACTION 1.0 AE_B B_I S_I T_I R_I AE_I K_I SH_I AH_I N_E
ABSTRACTIONS 1.0 AE_B B_I S_I T_I R_I AE_I K_I SH_I AH_I N_I Z_E
ABSURD 1.0 AH_B B_I S_I ER_I D_E
ABSURDITIES 1.0 AH_B B_I S_I ER_I D_I AH_I T_I IY_I Z_E
ABSURDITY 1.0 AH_B B_I S_I ER_I D_I AH_I T_I IY_E
ACCENT 1.0 AH_B K_I S_I EH_I N_I T_E
ACCENT 1.0 AE_B K_I S_I EH_I N_I T_E
ACCENTS 1.0 AE_B K_I S_I EH_I N_I T_I S_E
ACCEPTABLE 1.0 AE_B K_I S_I EH_I P_I T_I AH_I B_I AH_I L_E
ACCEPTABLE 1.0 AH_B K_I S_I EH_I P_I T_I AH_I B_I AH_I L_E
ACCEPTED 1.0 AE_B K_I S_I EH_I P_I T_I IH_I D_E
ACCEPTING 1.0 AE_B K_I S_I EH_I P_I T_I IH_I NG_E
ACCEPTING 1.0 AH_B K_I S_I EH_I P_I T_I IH_I NG_E
```

6.1 Introduction

The goal for this example is to develop a simple ASR system using the Kaldi toolkit . We hope to explore some potential issues and the general steps involved in the creation of a personalized ASR system.

In this example, we will use a series of audio files from various speakers, each containing an individual spoken digit from 0 to 9. Note that, in this example, a word is equivalent to a sentence and there is no sentence context (with only one word per file). This corpora is composed of several trials per speaker/digit. The goal is to train the system to recognize new audio files in which the speaker says a single digit from 0 to 9.

6.2 Resources

The tutorial in this example is based upon the “Kaldi for Dummies Tutorial” on the Kaldi site [17]. Our example goes slightly further in depth in some regions (especially the script results) and explores potential issues in the process.

This example requires audio data and, for the sake of time, we outsource the task of collection by using the audio files from the English-DATASET Provided.It is a large dataset with around 10000 audio files recorded by different speakers.

6.3 Preparing Audio Data

Audio samples were retrieved from the free-spoken-digit-dataset above, but it must be noted that there were certainly some issues with the given dataset in terms of incorporation into the Kaldi system.

Firstly, the data must be named in the fashion: speaker digit iteration.wav. This is done for sorting purposes – sorting by speaker id ends up being much more useful than sorting by digit. The data files in repository currently have the format digit speaker iteration.flac, so this format must be changed with a simple bash script that swaps the first element with the second element. Following this renaming process, we have to sort the audio files into speaker folders. This is accomplished using the *sort.sh* script in the Appendix. The resulting speaker folders must be placed in the data/test or data/train folders.

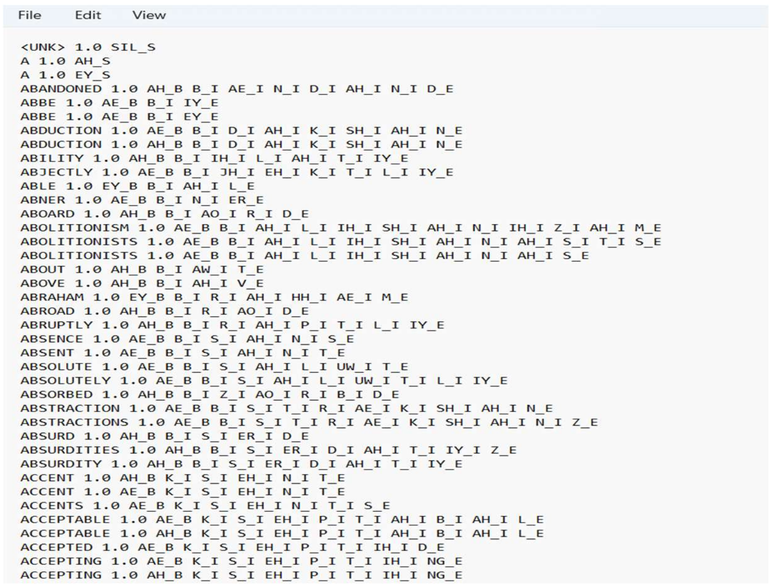
The next step is to generate the acoustic data. Luckily, because we have so few speakers and a very clear state-space of audio transcriptions, this data can be generated using a bash script (*acoustic.sh* in the Appendix). This script:

1. Organizes the data folder
2. Generates the *spk2gender* file for the test and train folders
3. Generates the *wav.scp* file for the test and train folders, matching utterance IDs to full paths in the directory
4. Generates the *text* file for the test and train folders, matching utterance ID to a transcription
5. Generates the *utt2spk* file for the test and train folders, matching utterance ID to speaker
6. Generates the corpus, *corpus.txt* (all possible text transcriptions in the ASR system)

By supplying the audio files and its accompanying acoustic data, we give the system a way to map new audio files to text transcriptions, given this particular system.

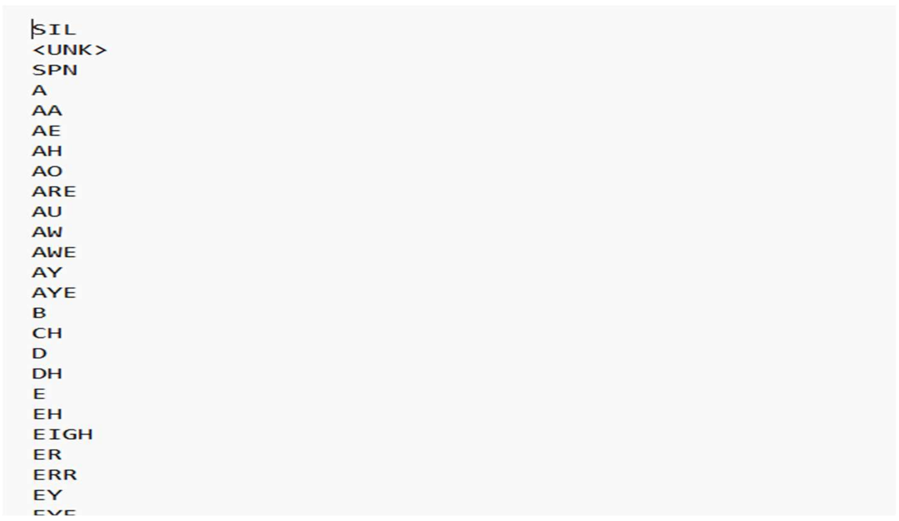
6.4 Language Data

The language data for this example can be manually entered. The lexicon (shown below) is a phonetic transcription of every possible word in the lexicon.



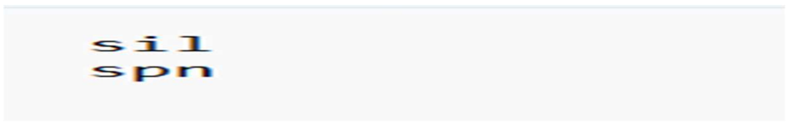
```
<UNK> 1.0 SIL_S
A 1.0 AH_S
A 1.0 EY_S
ABANDONED 1.0 AH_B B_I AE_I N_I D_I AH_I N_I D_E
ABBE 1.0 AE_B B_I IY_E
ABBE 1.0 AE_B B_I EY_E
ABDUCTION 1.0 AE_B B_I D_I AH_I K_I SH_I AH_I N_E
ABDUCTION 1.0 AH_B B_I D_I AH_I K_I SH_I AH_I N_E
ABILITY 1.0 AH_B B_I IH_I L_I AH_I T_I IY_E
ABJECTLY 1.0 AE_B B_I JH_I EH_I K_I T_I L_I IY_E
ABLE 1.0 EY_B B_I AH_I L_E
ABNER 1.0 AE_B B_I N_I ER_E
ABOARD 1.0 AH_B B_I AO_I R_I D_E
ABOLITIONISM 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I IH_I Z_I AH_I M_E
ABOLITIONISTS 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I AH_I S_I T_I S_E
ABOLITIONISTS 1.0 AE_B B_I AH_I L_I IH_I SH_I AH_I N_I AH_I S_E
ABOUT 1.0 AH_B B_I AW_I T_E
ABOVE 1.0 AH_B B_I AH_I V_E
ABRAHAM 1.0 EY_B B_I R_I AH_I HH_I AE_I M_E
ABROAD 1.0 AH_B B_I R_I AO_I D_E
ABRUPTLY 1.0 AH_B B_I R_I AH_I P_I T_I L_I IY_E
ABSENCE 1.0 AE_B B_I S_I AH_I N_I S_E
ABSENT 1.0 AE_B B_I S_I AH_I N_I T_E
ABSOLUTE 1.0 AE_B B_I S_I AH_I L_I UW_I T_E
ABSOLUTELY 1.0 AE_B B_I S_I AH_I L_I UW_I T_I L_I IY_E
ABSORBED 1.0 AH_B B_I Z_I AO_I R_I B_I D_E
ABSTRACTION 1.0 AE_B B_I S_I T_I R_I AE_I K_I SH_I AH_I N_E
ABSTRACTIONS 1.0 AE_B B_I S_I T_I R_I AE_I K_I SH_I AH_I N_I Z_E
ABSURD 1.0 AH_B B_I S_I ER_I D_E
ABSURDITIES 1.0 AH_B B_I S_I ER_I D_I AH_I T_I IY_I Z_E
ABSURDITY 1.0 AH_B B_I S_I ER_I D_I AH_I T_I IY_E
ACCENT 1.0 AH_B K_I S_I EH_I N_I T_E
ACCENT 1.0 AE_B K_I S_I EH_I N_I T_E
ACCENTS 1.0 AE_B K_I S_I EH_I N_I T_I S_E
ACCEPTABLE 1.0 AE_B K_I S_I EH_I P_I T_I AH_I B_I AH_I L_E
ACCEPTABLE 1.0 AH_B K_I S_I EH_I P_I T_I AH_I B_I AH_I L_E
ACCEPTED 1.0 AE_B K_I S_I EH_I P_I T_I IH_I D_E
ACCEPTING 1.0 AE_B K_I S_I EH_I P_I T_I IH_I NG_E
ACCEPTING 1.0 AH_B K_I S_I EH_I P_I T_I IH_I NG_E
```

The non-silence phones are those phones used that are not categorized as silence phones.



```
SIL
<UNK>
SPN
A
AA
AE
AH
AO
ARE
AU
AW
AWE
AY
AYE
B
CH
D
DH
E
EH
EIGH
ER
ERR
EY
```

The list of silence phones, used to represent silence or unknown sounds, is a short one (shown below):



```
sil
spn
```

In the optional silence phones text file, we just put *sil*.

6.5 SRI Language Model (SRILM)

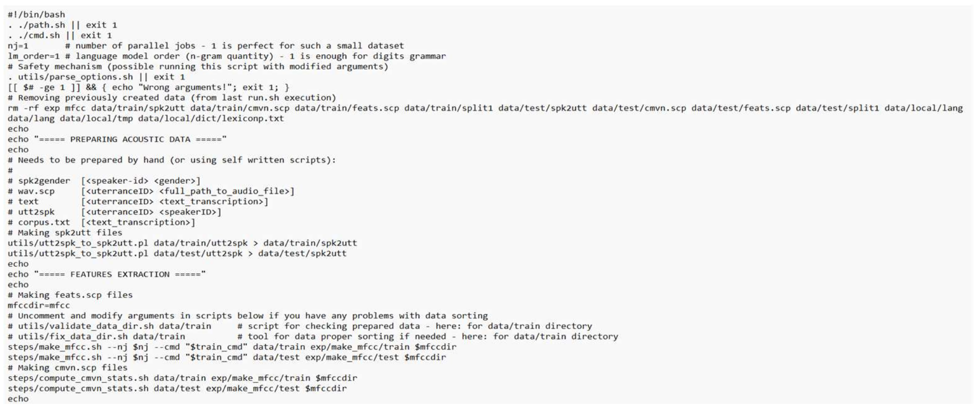
This particular example uses the SRI Language Model (SRILM) Toolkit [5]. SRILM is “a toolkit for building and applying statistical language models (LMs), primarily for use in speech recognition, statistical tagging and segmentation, and machine translation" [5]. Luckily, Kaldi has an *install srilm.sh* file in the extras folder, which can be run to bypass manual SRILM installation.

6.6 A Note on Sampling Rates

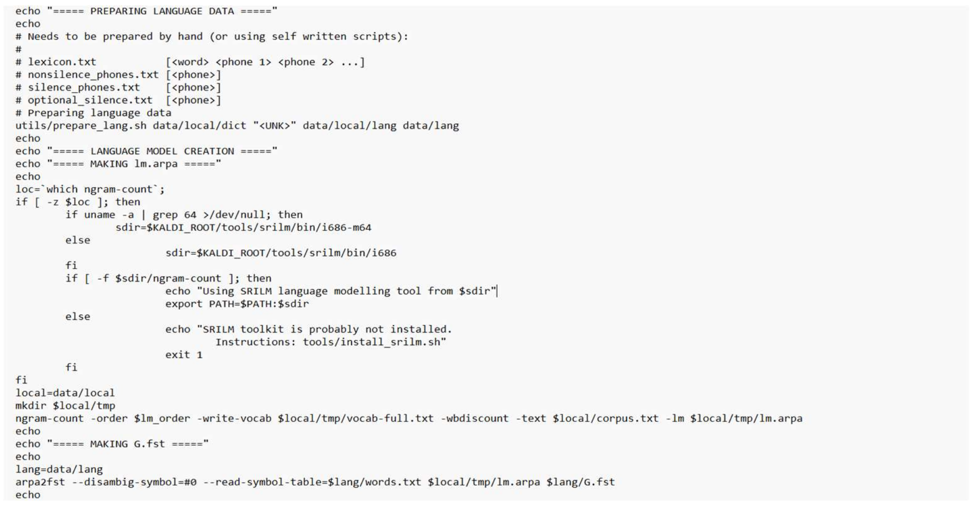
If you choose to use the same data as this example, you may have to re-sample the audio files. The language model used by SRILM in this example expects a 16kHz sampling rate. You can change the SRILM modeling sample rate, or you can re-sample the audio files with a script. See the Appendix for *resample.m*, a simple MATLAB script that re-samples the entire folder of digit audio files using piece-wise cubic hermite interpolation. As a note: It is important to be careful about resampling. Inserting a buffer "0" in between every data point in the audio, for instance, would allow the program to run, but would create interference around the Nyquist frequency and potentially produce erroneous results. Another option for resampling is SoX, the "Swiss Army Knife of Audio Manipulation," to re-sample the audio in the command line [6].

6.7 The "Run" Script

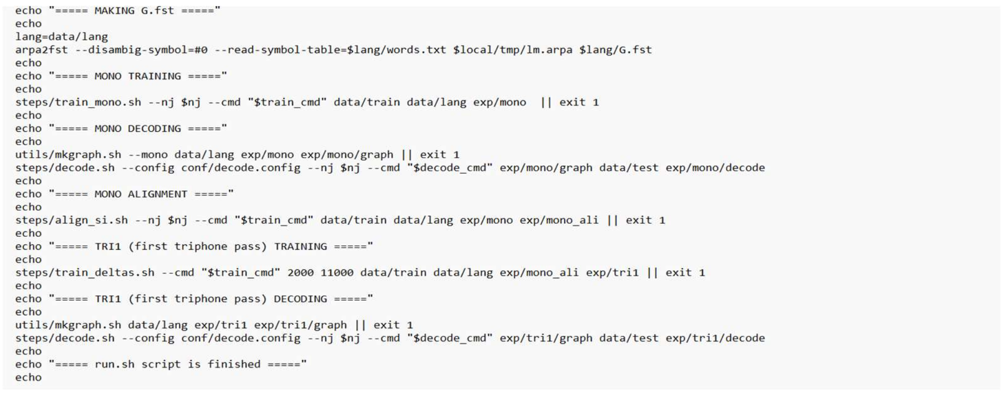
The *run.sh* script in each Kaldi example is used to execute all steps of the process, including data preparation, feature extraction, training and decoding. The script for *Digits* is relatively simple, and shows the general Kaldi process. Let's take a look at the general outline below (note that the unabridged version can be found in the appendix):



```
#!/bin/bash
. ./path.sh || exit 1
. ./cmd.sh || exit 1
nj=1       # number of parallel jobs - 1 is perfect for such a small dataset
lm_order=1 # language model order (n-gram quantity) - 1 is enough for digits grammar
# Safety mechanism (possible running this script with modified arguments)
. utils/parse_options.sh || exit 1
[[ $# -ge 1 ]] && { echo "Wrong arguments!"; exit 1; }
# Removing previously created data (from last run.sh execution)
rm -rf exp mfcc data/train/spk2utt data/train/cmvn.scp data/train/feats.scp data/train/split1 data/test/spk2utt data/test/cmvn.scp data/test/feats.scp data/test/split1 data/local/lang
data/lang data/local/tmp data/local/dict/lexiconp.txt
echo
echo "===== PREPARING ACOUSTIC DATA ====="
echo
# Needs to be prepared by hand (or using self written scripts):
#
# spk2gender  [<speaker-id> <gender>]
# wav.scp     [<uterranceID> <full_path_to_audio_file>]
# text        [<uterranceID> <text_transcription>]
# utt2spk     [<uterranceID> <speakerID>]
# corpus.txt  [<text_transcription>]
# Making spk2utt files
utils/utt2spk_to_spk2utt.pl data/train/utt2spk > data/train/spk2utt
utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt
echo
echo "===== FEATURES EXTRACTION ====="
echo
# Making feats.scp files
mfccdir=mfcc
# Uncomment and modify arguments in scripts below if you have any problems with data sorting
# utils/validate_data_dir.sh data/train     # script for checking prepared data - here: for data/train directory
# utils/fix_data_dir.sh data/train          # tool for data proper sorting if needed - here: for data/train directory
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/train exp/make_mfcc/train $mfccdir
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/test exp/make_mfcc/test $mfccdir
# Making cmvn.scp files
steps/compute_cmvn_stats.sh data/train exp/make_mfcc/train $mfccdir
steps/compute_cmvn_stats.sh data/test exp/make_mfcc/test $mfccdir
echo
```



```
echo "===== PREPARING LANGUAGE DATA ====="
echo
# Needs to be prepared by hand (or using self written scripts):
#
# lexicon.txt           [<word> <phone 1> <phone 2> ...]
# nonsilence_phones.txt [<phone>]
# silence_phones.txt    [<phone>]
# optional_silence.txt  [<phone>]
# Preparing language data
utils/prepare_lang.sh data/local/dict "<UNK>" data/local/lang data/lang
echo
echo "===== LANGUAGE MODEL CREATION ====="
echo "===== MAKING lm.arpa ====="
echo
loc=`which ngram-count`;
if [ -z $loc ]; then
        if uname -a | grep 64 >/dev/null; then
                sdir=$KALDI_ROOT/tools/srilm/bin/i686-m64
        else
                        sdir=$KALDI_ROOT/tools/srilm/bin/i686
        fi
        if [ -f $sdir/ngram-count ]; then
                        echo "Using SRILM language modelling tool from $sdir"
                        export PATH=$PATH:$sdir
        else
                        echo "SRILM toolkit is probably not installed.
                                Instructions: tools/install_srilm.sh"
                        exit 1
        fi
fi
local=data/local
mkdir $local/tmp
ngram-count -order $lm_order -write-vocab $local/tmp/vocab-full.txt -wbdiscount -text $local/corpus.txt -lm $local/tmp/lm.arpa
echo
echo "===== MAKING G.fst ====="
echo
lang=data/lang
arpa2fst --disambig-symbol=#0 --read-symbol-table=$lang/words.txt $local/tmp/lm.arpa $lang/G.fst
echo
```



```
echo "===== MAKING G.fst ====="
echo
lang=data/lang
arpa2fst --disambig-symbol=#0 --read-symbol-table=$lang/words.txt $local/tmp/lm.arpa $lang/G.fst
echo
echo "===== MONO TRAINING ====="
echo
steps/train_mono.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/mono  || exit 1
echo
echo "===== MONO DECODING ====="
echo
utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd" exp/mono/graph data/test exp/mono/decode
echo
echo "===== MONO ALIGNMENT ====="
echo
steps/align_si.sh --nj $nj --cmd "$train_cmd" data/train data/lang exp/mono exp/mono_ali || exit 1
echo
echo "===== TRI1 (first triphone pass) TRAINING ====="
echo
steps/train_deltas.sh --cmd "$train_cmd" 2000 11000 data/train data/lang exp/mono_ali exp/tri1 || exit 1
echo
echo "===== TRI1 (first triphone pass) DECODING ====="
echo
utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1
steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd" exp/tri1/graph data/test exp/tri1/decode
echo
echo "===== run.sh script is finished ====="
echo
```

This process can be broken down into a series of steps, starting at data preparation and continuing to training and decoding:

1. **Preparing acoustic data** (using the audio files)
2. **MFCC feature extraction** using train and test data
3. **Preparing language data** (relating to the possible phones seen and the breakdown of words into phones)
4. **Language model creation** (here, using SRILM)

   Making *lm.arpa* (the language model, as an ARPA file[1]) Making *G.fst* (converted from *lm.arpa* to a FST file[2])
5. Monophone Speech Recognition: *does not* include any contextual information about the preceding or following phone [10]

   Training

   Decoding

   Alignment
6. Triphone Speech Recognition: *does* include any contextual information about the preceding or following phone

   Training (first pass)

   Decoding (first pass)

We can see a sample output in the Appendix under *Digits run.sh Output*. It is too long to include here.

[1] An ARPA file uses log probabilities to convey phrase probabilities [14]

[2] An FST file is a binary representation of the finite state transducer/acceptor [20]

6.8 Results

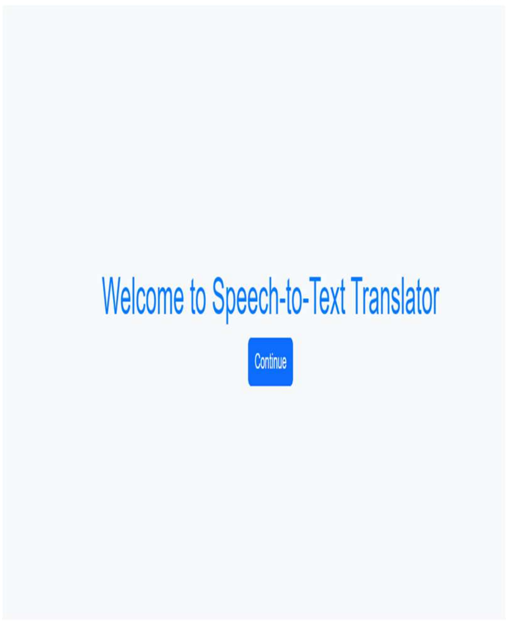
6.8.1 Decoding Logs

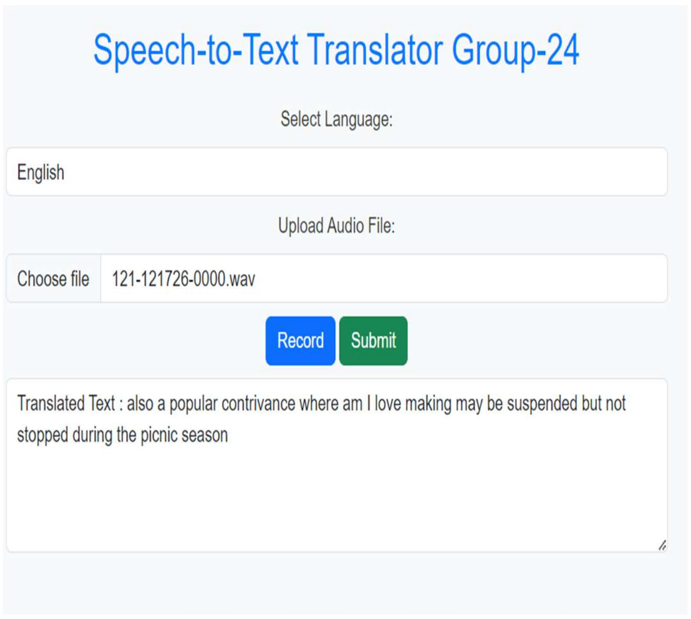
One easy way to observe the script's functionality is to look at the decoding logs generated via the script. In the logs, we can see the utterance ID paired to the predicted transcription (seen in Figure 5). In our example log, we can see successful transcriptions (in green) and failed transcription (in red).

6.8.2 Word Error Rates

Another way to assess the script results is to look at the resulting *Word Error Rates.* During the monophone and triphone decoding phases, the script generates a series of *Word Error Rates* (WER). The WER is used to measure the accuracy of the ASR system. The WER is calculated as the minimum edit distance between the output of the ASR system and the reference transcriptions. The relevant edit operations are substitution, deletion and insertion [16]. The expression for WER is shown below in Equation 1.

$$WER = 100 * \frac{min\_dist(decoded(a), t, edit\_op = sub, del, ins)}{num\_words(t)}$$





English

121-121726-0000.wav

Translated Text : also a popular contrivance where am I love making may be suspended but not stopped during the picnic season

Fig 6.1-Welcome Page

Fig 6.2 - Home Page

10 Conclusion

The ASR model was trained using the Kaldi framework, leveraging its powerful capabilities for acoustic and language model training. The use of state-of-the-art techniques contributed to the model's robustness and accuracy. The successful deployment of the ASR model showcases its practical applicability. Integration into the deployment environment was seamless, and the model demonstrated consistent performance across various scenarios. Rigorous evaluation of the ASR model yielded positive results. Key performance metrics, such as Word Error Rate (WER) or accuracy, demonstrated the effectiveness of the model in accurately transcribing spoken English. The deployed ASR system exhibited scalability and efficiency, handling a diverse range of input scenarios and maintaining responsiveness even under varying workloads. While the current implementation has proven successful, ongoing efforts may focus on [potential areas for improvement/enhancement]. This could include exploring new training data, fine-tuning parameters, or considering additional features to enhance model performance. In summary, the completion of this project marks a significant milestone in the development of an effective ASR solution using Kaldi. The successful deployment and positive outcomes underscore the potential for real-world applications, and the insights gained will inform future developments in the field.

References

[1] ACG's Sun Grid Engine (SGE) Cluster. University of Pennsylvania.

[2] CMU Census Database, 1991. Carnegie Mellon University.

[3] VoxForge: Open Source Acoustic Model and Audio Transcriptions, 20062017. VoxForge.

[4] Sun Cluster Data Service for Sun Grid Engine Guide for Solaris OS, 2010. Oracle.

[5] SRILM - The SRI Language Modeling Toolkit, 2011. SRI International.

[6] SoX - Sound eXchange, 2015. PmWiki.

[7] Tanel Alum¨ae. Kaldi GStreamer server, 2017.

[8] Madeline Briere. Speech Processing and Recognition, 2017.

[9] Berlin Chen. An Introduction to the Kaldi Speech Recognition Toolkit, 2014. National Taiwan Normal University.

[10] Eleanor Chodroff. Corpus Phonetics Tutorial: Kaldi, 2017. Northwestern University.

[11] Gruhn et. al. Automatic Speech Recognition, 2011. Statistical Pronunciation Modeling for Non-Native Speech Processing.

[12] Mirko Hannemann. Weighted Finite State Transducers in Automatic Speech Recognition, 2013. Computer Science Department at Brandeis.

[13] Zohar Jackson. Free Spoken Digit Dataset (FSDD), 2017.

[14] Duane Johnson. Understanding ARPA and Language Models, 2014. WordTree.

[15] Sundar Pichai. Google's I/O Developer Conference, 2017.

[16]Ondˇrej Pl´atek. Automatic speech recognition using Kaldi. Institute of Formal and Applied Linguistics.

[17] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi Speech Recognition Toolkit, 2011.

[18] George Saon. Recent Advances in Conversational Speech Recognition, 2016. IBM.

[19] Sam Skipsey. Reresolve hostname failed: can't resolve hostname with SGE 6.1, 2008. University of Liverpool.

[20] Lyndon White. Kaldi-notes: Some notes on Kaldi. University of Western Australia.